ISBN 82-553-0346-4

Mathematics
No 6 - May 1978

FINITE ALGORITHMIC PROCEDURES AND INDUCTIVE DEFINABILITY

by

J. Moldestad, V. Stoltenberg-Hansen
and J.V. Tucker

PREPRINT SERIES - Matematisk institutt, Universitetet i Oslo

Associated with any relational structure A there are a number of kinds of functions on A distinguished by their simple combinatorial relationship with the basic operations and relations of A, specifically by their finite mode of computation over the domain from its given relational structure. Along with A imagine the family of all A-register machines each member of which can carry some specific finite number of elements of A, perform the basic operations and decide the basic relations on these elements, and manage a few simple manipulations and decisions such as to replace the contents of one register by those of another and to tell when two registers contain the same elements. To use such a machine to compute a partial function $f: A^m \to A$ is to write down the familar finite programme of instructions referring to these possible activities of the machine and containing information to stop in certain circumstances: given $\underline{a} \in A^m$ as an input the programme determines a pattern of behaviour by the machine which ends if and only if $f(\underline{a})$ is defined and then with $f(\underline{a})$ in its output register.

Such a programme is called a finite algorithmic procedure, a fap, for short. A function $f: A^m \to A$ is fap-computable iff there is a fap which, together with an appropriate A-register machine, will compute the value $f(\underline{a})$ from each argument input $\underline{a}$. The set of all fap-computable functions $A^m \to A$ in their entirity for each m is denoted FAP(A).

Extensions of this first class of computable functions on A are obtained by refining the capabilities of the computing devices: allowing certain enlargements of the machine's storage facilities, or by allowing subcomputations on the natural numbers ω, or by arranging both. An extension of the first kind is particularly important here.

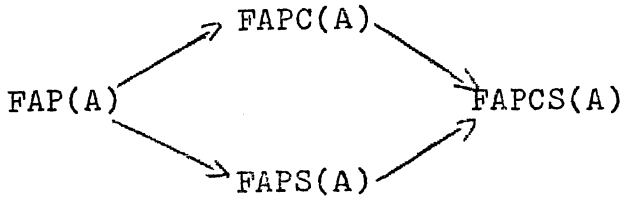
An A-register machine with a stack has the further facility of a special register in which the entire contents of the ordinary registers can be stored, along with one of a finite number of prescribed markers, at various points in the course of a calculation, the intention being to enlarge the number and complexity of subcomputations. With the new instructions a programme for these machines is called a finite algorithmic procedure with stacking, or a fapS, and the class of fapS-computable functions they define we denote FAPS(A). (These classes are properly defined in section one.)

Ancillary to this paper, but germane to its sequel, are extensions involving arithmetic. An A-register machine with counting registers has a finite number of numerical registers adjoined with the new operations of being able to add or subtract 1 from the contents of any counting register and to decide when two registers contain the same number and so on. Programmes appropriate for these machines are called finite algorithmic procedures with counting, and the class of all fapC-computable functions they define is denoted FAPC(A). Combining stacking and counting in register machines leads to the class FAPCS(A). (These classes are properly defined in [13].)

H. Friedman first considered the functions FAP(A) and FAPC(A) in [7] and so invented a most plausible conception of how to analyse computing in an abstract setting. The classes FAPS(A) and FAPCS(A) are our own invention but have been identified and studied in variant forms by R.C. Constable & D. Gries [1] in the former case and J.C. Shepherdson [18] in the latter, exact details are included in [13].

The generalised recursion theory of these machine-theoretic functions is the subject of this paper and its companion [13]. We

shall show that, in general, the four kinds of computing power differ from oneanother, the inclusions being



with stacking and counting incomparable and, moreover, that each type of computing underlies important theoretical ideas in abstracting recursion theory from ω to a relational structure A.

In this paper the classes FAP(A) and FAPS(A) are characterised by function-theoretical means of generating functions from the basic operations and relations of A. These methods are derived from the unpublished work of R.A. Platek on inductive definitions [17]. Among many things, Platek discovered the abstract nature of recursion over arbitrary finite types showing that the theory of recursion presented in Kleene's [8,9] could be given over any set with some primitive structure; an account of this work of Platek is included in Moldestad's [12]. In section two we discuss simple abstract recursion on a relational structure A, with finitely many operations and relations, in order to define the class of recursive or, as we prefer to say, inductive functions on A, denoted Ind(A), and a subclass DInd(A) of directly inductive functions on A. In sections three and four respectively we prove

Theorem 1 $\quad DInd(A) = FAP(A)$

Theorem 2 $\quad Ind(A) = FAPS(A)$

In the companion paper we examine machine computable functions from the point of view of the axiomatic analysis of recursion theory,

that of Moschovakis [14,15,16] and, in particular, Fenstad [3,4,6]. There the central classes prove to be FAPC(A) and FAPCS(A) for these, it is shown, characterise minimal computing strengths on A necessary to generate workable computation theories (in the large). To that paper we postpone the discussion of computing with counting and so, too, the division of the classes in the diagram above and the corollaries of the theorems here involving them.

The results of this paper, and those of its sequel, are pertinent to considerations of strategies for perfecting a general recursion/computability theory in the situation of an indefinite structure for the roles of arithmetic, and of pairing (though not that of search operators) together with the precise relationships between computing commitments is exactly determined; references in mind are Feferman [2] and Fenstad [5,6].

But it is not to these concerns of <u>theoria</u> that these articles are committed exclusively, rather to another circle of ideas about generalised computing which aims to create a theory of computing in algebraic systems which appeals to an algebraist's turn of mind and which can be used in algebraic investigations where questions of definability, constructiveness and complexity are involved. For the recursion theorist, however, from the fact that the delicate algebraic properties of the given system A materially influences the structure of the computing theories derived over A there is the promise of a rich and subtle <u>praxis</u> for generalised recursion relevant to its most elementary levels. Should this interest the reader, we refer him or her to the introductory paper [19].

One of us - Tucker - wishes to acknowledge the indispensible support of a fellowship from the European Programme of the Royal Society, London.

1. Finite Algorithmic Procedures

First, the few ideas from the theory of universal algebras used in this paper and in its companion are to be found in, for example, Mal'cev's book [10]. The relational structures considered are of the form $A = (A; \sigma_1, \cdots, \sigma_l, S_1, \cdots, S_s)$ where the operations and relations are finitary and need not be total.

If X, Y are non-empty sets then by $P(X,Y)$ we denote the set of all partial functions $X \to Y$; the domain of definition of $f \in P(X,Y)$ is written $\mathrm{dom}(f)$.

An essential reference on faps is Friedman's article [7]. Let us take as understood the concept of an A-register machine with n registers M_A^n. Programmes for such machines are written in the following language.

Constants are $\emptyset$ for the empty register, H for halt. Variables are $r_0, r_1, r_2, \cdots$ for algebra registers. Function symbols and relative symbols are those used for the species of the relational structure A.

A programme or finite algorithmic procedure P is an ordered finite list of instructions $(I_1, \cdots, I_k)$ where instructions are of two kinds.

The operational instructions which manipulate elements of A are

$r_\mu := \sigma(r_{\lambda_1}, \cdots, r_{\lambda_m})$ meaning "apply the m-ary operation σ to the contents of registers $r_{\lambda_1}, \cdots, r_{\lambda_m}$ and replace the content of register r_μ by this value."

$r_\mu := r_\lambda$ meaning "replace the content of register r_μ with that of r_λ."

$r_\mu := \emptyset$ meaning "empty register r_μ."

H meaning "stop."

The conditional instructions which determine the order of implementing instructions are

if $S(r_{\lambda_1},\cdots,r_{\lambda_m})$ then i else j meaning "if the n-ary relation S is true of the contents of $r_{\lambda_1},\cdots,r_{\lambda_m}$ then the next instructions is I_i otherwise it is I_j."

if $r_\mu = r_\lambda$ then i else j meaning "if registers r_μ and r_λ contain the same element then the next instruction is I_i otherwise it is I_j."

if $r_\mu = \emptyset$ then i else j meaning "if register r_μ is empty then the next instruction is I_i otherwise it is I_j."

The special conditional relation $r_\mu = r_\mu$ gives an instruction abbreviated goto i.

The instructions making up the fap P are executed on a machine M in the order in which they are given except where a conditional instruction directs otherwise. By convention, a fap P always involves an initial segment of the register variables $r_0, r_1, \cdots, r_n$ where the first few registers $r_1,\cdots,r_m$ are reserved as input registers and r_0 as output register; the remaining registers mentioned in P are called working registers. Thus a given fap P with n input registers and n-m-1 working registers, together with an appropriate machine M_A defines a partial function $A^m \to A$ in the obvious way: load the argument $\underline{a} \in A^m$ into the

input registers and start the programme P, if the machine halts and the output register r_0 is not empty, then the value of the function $P(\underline{a})$ is defined to be the element in r_0, else no value of the function on $\underline{a}$ is defined.

$f \in P(A^m,A)$ is <u>fap-computable</u> iff there exists a fap P and a machine M such that for each $\underline{a} \in A^m$, $f(\underline{a}) = P(\underline{a})$.

Actually, there are a number of conditions on programmes which limit further the set of programmes without affecting the class of functions computed. For example, we can insist that there is always at least one halt instruction or, indeed, that there is exactly one and that it is the final instruction of the programme. In our work with fap computations no such hypotheses are in operation. And, finally, notice that the instructions involving empty registers do not affect the class of fap-definable functions over structures with two or more elements, for this reason we ignore them in the arguments which follow.

To understand the nature of an A-register machine with n registers and a stack $M_A^{S,n}$ it suffices to consider its instructions. The basic device M_A^n is extended by a stack register where the contents of the n original registers can be stored as an n-tuple with one of a finite number of labels.

Append to the syntax for faps the new constants $1,2,\cdots$ for markers and the variable s for stack register. The new operational instructions are

$s := (i,r_0,\cdots,r_{n-1})$ meaning "place a copy of the contents of the registers $r_0,\cdots,r_{n-1}$ as an n-tuple in the stack register together with the marker i."

restore $(r_0, r_1, \cdots, r_{j-1}, r_{j+1}, \cdots, r_{n-1})$ meaning "replace the contents of the registers $r_0, r_1, \cdots, r_{j-1}, r_{j+1}, \cdots, r_n$ by those of the last, or topmost, n-tuple placed in the stack."

The new conditional instruction is

if $S = \emptyset$ then i else j

and it takes its natural meaning.

In writing fapS's it is necessary to regulate how these new instructions appear in the basic faps through devising stacking blocks of instructions. A stacking block is a sequence of consequent instructions of the following form

```
s: = (i; r_0,···,r_{n-1})
I_1
.
.
.
I_1
goto k
*: r_j: = r_0
restore (r_0,r_1,···,r_{j-1},r_{j+1},···,r_{n-1})
```

The marker i is unique to the block in any programme in which that block appears. The $I_1, \cdots, I_1$ are ordinary fap operational instructions referred to as (re-)loading instructions. The instruction I_k has a special rôle in the operation of the block, it is called the return instruction of the block and it must be either an ordinary fap instruction outside all the blocks in the programme or it is the

first instruction of any block in the programme. The instruction (informally) prefixed by an asterisk is called the exit instruction.

The halt instruction also takes on the form of a block

I_i	if s = ∅ then i+1 else i+2
I_{i+1}	H
I_{i+2}	goto (exit instruction of the block whose marker is topmost in the stack).

This halting block we abbreviate

if s = ∅ then H else * .

The new instructions involving the stack may only occur in a stacking block or a halting block.

Further conventions we operate are: from an ordinary fap conditional instruction one may not enter a block except by way of its first instruction. The last instruction of a programme, if it is not a conditional, is the last instruction of a block.

So a finite algorithmic procedure with stacking is defined to be a programme of instructions satisfying the conditions and conventions described. A given fapS P, together with a machine M, defines a partial function over A in the obvious way and $f \in P(A^m, A)$ is said to be fapS-computable iff there exists an appropriate faps P and machine M to compute it.

We open section 4 with an example of an fapS.

In working with programmes we shall often corrupt the formal language and instruction forms with informal descriptions where this simplifies our exposition.

2. Inductive Definability.

The inductively definable functions on A are created from the system's operations and relations - presented in the form of definition-by-cases functions - by means of composition and taking fixed-points of certain specially constructed monotone functionals. Given Kleene's revision of recursion, this class Ind(A) is a natural candidate for that of the recursive functions on A. We propose to give an entirely syntactic definition of Ind(A) but will first consider it in a rather algebraic way.

In working with partial functions on A it is convenient to replace A by A_u being A with the symbol u for undefined adjoined; operations and relations take their obvious definitions on A_u: the value of a function on an argument involving u being u. We omit the subscript as there are no opportunities for confusion.

Consider simultaneously partial functions of all arguments over A, $P(A) = \bigcup_{m\in\omega} P(A^m,A)$, in which we specify a basic family of functions and on which we shall ultimately define two generating processes. The initial functions are these

i. For each m, the projection functions $U_i^m(a_1,\cdots,a_m) = a_i$, $1 \leq i \leq m$ from $P(A^m,A)$.

ii. If σ is an m-ary operation of A then σ from $P(A^m,A)$.

iii. If S is an m-ary relation of A then

$$DC_S(a_1,\cdots,a_m,x,y) = x \quad \text{if} \quad S(a_1,\cdots,a_m)$$
$$= y \quad \text{if} \quad S(a_1,\cdots,a_m)$$

from $P(A^{m+2},A)$.

iv. u_m the nowhere defined function from $P(A^m,A)$.

The operations on $P(A)$ are general compositions $C^{n,m}: P(A^n,A) \times P(A^m,A)^n \longrightarrow P(A^m,A)$ defined $C^{n,m}(f,g_1,\cdots,g_n)(\underline{a}) = f(g_1(\underline{a}),\cdots,g_n(\underline{a}))$ and more complicated operations involving fixed-points which we now begin to describe.

Let X,Y be non-empty sets. If $f,g \in P(X,Y)$ then f is a subfunction of g, $f \leq g$, iff $\mathrm{dom}(f) \subset \mathrm{dom}(g)$ and for each $x \in \mathrm{dom}(f)$, $f(x) = g(x)$.

A map $\psi: P(X,Y) \to P(X,Y)$ is monotonic iff for each $f,g \in P(X,Y)$, if $f \leq g$ then $\psi(f) \leq \psi(g)$.

And ψ is continuous iff for each $f \in P(X,Y)$ and any $y_1,\cdots,y_n \in X$ there exist $x_1,\cdots,x_m \in X$ such that if $g(x_i) = f(x_i)$, $1 \leq i \leq n$ then $\psi(g)(y_i) = \psi(f)(y_i)$ for $1 \leq i \leq n$.

The set of all continuous and monotonic maps $P(X,Y) \to P(X,Y)$ we denote $CM(P(X,Y),P(X,Y))$; it is closed under composition.

2.1 Least Fixed-Point Theorem

A continuous monotonic function $\psi: P(X,Y) \to P(X,Y)$ has a unique least fixed-point ψ^* and, moreover, $\psi^* = \operatorname{lub}_{n\in\omega} \psi^n(u)$.

Proof: Define the countable sequence $f_0 = u$, $f_{n+1} = \psi(f_n)$. It is easy to see that for each n, $f_n \leq f_{n+1}$. By induction on n: it is true for $n = 0$ as u is a subfunction of any function. If $f_n \leq f_{n+1}$, then $\psi(f_n) \leq \psi(f_{n+1})$ as ψ is monotonic, but this is the relation $f_{n+1} \leq f_{n+2}$. So set $f = \bigcup_{n\in\omega} f_n$, the least upper bound of the $\psi^n(u)$.

We claim that $\psi(f) = f$ and that if $\psi(g) = g$, then $f \leq g$.

$f \leq \psi(f)$ follows from monotonicity: for any n, $f_{n-1} \leq f$ thus $\psi(f_{n-1}) \leq \psi(f)$ and $f_n \leq \psi(f)$. So $f \leq \psi(f)$.

$\psi(f) \leq f$ requires continuity: let $x \in dom(f)$, by continuity of ψ there exist $x_1,\ldots,x_m \in X$ such that $g(x_i) = f(x_i)$, $1 \leq i \leq m$, entails $\psi(g)(x) = \psi(f)(x)$, choose $g = f\upharpoonright\{x_1,\ldots,x_m\}$. Now $g \leq f_n$ for some n and $\psi(g) \leq \psi(f_n) = f_{n+1} \leq f$.

Finally, to show that if $\psi(g) = g$, then for each n, $f_n \leq g$ we use induction. The statement is obviously true for $n = 0$. If $f_n \leq g$ then $\psi(f_n) \leq \psi(g)$ which is $f_{n+1} \leq g$.

Q.E.D.

Thus for each m there is a <u>fixed-point operator</u> FP^m: $CM(P(A^m,A),P(A^m,A)) \longrightarrow P(A^m,A)$ defined $FP^m(\psi) = \psi^*$ inductively and constructively in this proof. For the existence and inductive character of a least fixed-point the hypothesis of continuity is immaterial, it is included because constructivity is required; a useful reference for fixed-points is Manna & Shamir's [11]. In these fixed-point operators is the essence of recursion and to complete the definition of Ind(A) we have only to explain the construction of appropriate continuous, monotonic functionals from given partial functions. To present this as a genuine algebraic operation on P(A) requires a substantial digression on the algebraic structure of P(A), as we intend to follow Platek's equational calculus definition of Ind(A) this we do not persue. Actually, P(A) is rich both in structure - P(A) is a complete semilattice under $\leq$, and a topological algebra under $C^{n,m}$ together with certain other operations, hence the terminology of "continuous" above - and in distinguished classes of functions - for example, the subalgebra of $(P(A); C^{n,m}, n,m \in \omega)$ generated by functions in (i) - (iii) is an important extention of the polynomials over A; for information on this algebraic point of view see [20].

For the syntactic definition naturally we work with respect to the species of A with standard operation and relation notation $\sigma_1,\cdots,\sigma_l$ and $S_1,\cdots,S_s$. The terms required are defined inductively and solely by the following clauses:

i. the algebra element indeterminates $X = \{x_1,x_2,\cdots\}$ are terms of type 0;

ii. for each m, the m-ary partial function indeterminates $P^m = \{p_1^m,p_2^m,\cdots\}$ are terms of type 1.m;

iii. for each m-ary operation σ the function symbol $\underline{\sigma}$ is a term of type 1.m;

iv. For each m-ary relation S the function symbol $\underline{DC}_S$ is a term of type 1.m+2;

v. $\underline{u}$ is a term of type 0;

vi. if T is a term of type 1.m and $t_1,\cdots,t_m$ are terms of type 0 then $T(t_1,\cdots,t_m)$ is a term of type 0;

vii. if t is a term of type 0 then $FP[\lambda p_i^m,y_1,\cdots,y_m.t]$ is a term of type 1.m; here $y_1,\cdots,y_m$ are algebra indeterminates which along with p_i^m are closed in the whole term.

Let $P = \bigcup_{m\in\omega} P^m$. Let T_0 be the set of terms of type 0 and T_1 the set of terms of type 1 so called <u>algebra terms</u> and <u>function terms</u> respectively.

It is intuitively clear how this syntax is used to define the recursive functions: a partial function $f: A^m \to A$ will be inductively definable iff there is an algebra term $t(y_1,\cdots,y_m)$ with $y_1,\cdots,y_m$ its only free variables such that for all $a_1,\cdots,a_m \in A$,

$f(a_1,\cdots,a_m) = t(a_1,\cdots,a_m)$. Here is an outline of the proper mechanism involving valuation functions which interpret the terms in our algebra.

A <u>valuation function</u> $V: T_0 \to A$ and $V: T_1 \to P(A)$ is determined by its values on the indeterminates X and P extending to all terms as follows:

For each operation symbol $\underline{\sigma}$, $V(\underline{\sigma}) = \sigma$; for each relation symbol $\underline{S}$, $V(DC_{\underline{S}}) = DC_S$; $V(\underline{u}) = u$. And,

$$V(T(t_1,\cdots,t_m)) = V(T)(V(t_1),\cdots,V(t_m))$$

$$V(FP[\lambda p_i^m, y_1,\cdots,y_m.t]) = FP^m(\psi_{V,t})$$

wherein $\psi_{V,t}$ is the functional inductively defined by $\psi_{V,t}(f)(a_1,\cdots,a_m) = V'(t)$ where $V'(t)$ is constructed from V except on $p_i^m, y_1,\cdots,y_m$ where $V'(p_i^m) = f$ and $V'(y_i) = a_i$ for $1 \leq i \leq m$. This extension is routine except in the last case. It must be shown that

2.2 <u>Lemma</u>. Let t be an algebra term with free function variables among $y_1,\cdots,y_n$. For any valuation function V the functional $\psi_{V,t}$ is continuous and monotonic.

<u>Proof</u>: This we merely sketch, it is by induction on the complexity of the term t. Cases for t presented by clauses (i) - (vi) are routine and include the base steps of the induction. Consider case (vii) where t is $FP^k[\lambda p_i^k, z_1,\cdots,z_k.t_0](t_1,\cdots,t_k)$. From the induction hypothesis define continuous and monotonic maps $\Psi,\psi_1,\cdots,\psi_k$ by

$\Psi(f_1,\cdots,f_m,g)(a_1,\cdots,a_n,b_1,\cdots,b_k)$ = value of t_0 with substitutions $y_j = a_j (1 \leq j \leq n)$, $z_j = b_j (1 \leq j \leq k)$, $p_j = f_j (1 \leq j \leq n)$, $p^k = g$.

And, for $1 \leq i \leq k$, $\psi_i(f_1,\cdots,f_m)(a_1,\cdots,a_n)$ = value of t_i with

$$y_j = a_j \ (1 \leq j \leq n),$$
$$p_j = f_j \ (1 \leq j \leq m).$$

Set $f = (f_1,\cdots,f_m)$ and $\Psi_{f,a}(g)(\underline{b}) = \Psi(f,g)(a,b)$ for $\underline{a} \in A^n$, $\underline{b} \in A^k$

so that $\psi_{V,t}(f)(\underline{a}) = FP^k[\Psi_{f,a}](\psi_1(f)(\underline{a}),\cdots,\psi_k(f)(\underline{a}))$

$$= \operatorname*{lub}_{j\in\omega}(\Psi^j_{f,a}(u))(\psi_1(f)(\underline{a}),\cdots,\psi_k(f)(\underline{a}))$$

by the least Fixed-point Theorem; rewriting as, say,

$$\psi_{V,t}(f)(\underline{a}) = \Phi(f,\underline{a})(\psi_1(f)(\underline{a}),\cdots,\psi_k(f)(\underline{a})),$$

it is sufficient to show that for each a the functional $f \to \Phi(f,\underline{a})$ is continuous and monotonic as the $\psi_1,\cdots,\psi_k$ are by hypothesis. This follows routinely from the claim that for each j, $\Phi^j(f,\underline{a})(\underline{b}) = \Psi^j_{f,a}(u)(\underline{b})$ is continuous and monotonic which is proved by induction on j and is omitted.

Q.E.D.

A partial function $f: A^m \to A$ is <u>inductively definable</u> iff there is an algebra term t containing free variables $y_1,\cdots,y_m$ such that for each $\underline{a} \in A^m$ and any valuation V wherein $V(y_i) = a_i$ then $f(\underline{a}) = V(t)$; in such circumstanees f is said to be <u>defined</u> by t.

The directly inductive functions DInd(A) are obtained from a subset of T_0.

Let t be an algebra term with p a function variable free in t. p is said to <u>occur in a conditional place</u> in t iff there is a subterm $\underline{DC_S}(t_1,\cdots,t_m,t_{m+1},t_{m+2})$ such that p occurs in one (or more) of the t_i, $1 \leq i \leq m$.

And p is said to <u>occur in the scope of a function term</u> in t

iff p occurs in one of $t_1,\cdots,t_m$ and $T(t_1,\cdots,t_m)$ is a subterm of t where T is any of (ii), (iii) and (vii).

An algebra term t is said to be <u>direct</u> iff for each subterm t_0 all function variables which are free in t_0 do not occur in a conditional place in t_0 nor in the scope of a function term in t_0.

A partial function $f: A^m \to A$ is <u>directly inductively definable</u> iff it can be defined by a direct term.

Here are some examples we encounter later.

$$FP[\lambda p',y.\ \underline{DC_S}(y,y,\underline{DC_S}(\underline{\sigma}(y),y,p'(\underline{\sigma}(y))))](x)$$

is a direct term, and

$$FP[\lambda p',y.\ \underline{DC_S}(y,y,\underline{DC_S}(\underline{\sigma_1}(y),y,\underline{\sigma_2}(p'(\underline{\sigma_1}(y)),\underline{\sigma_1}(y))))](x)$$

is a term which is not direct.

3. Proof of Theorem 1

3.1. Proposition. If a function is fap-computable then it is directly inductively definable.

Proof: Suppose f is defined by a programme P with input registers $r_1,\ldots,r_m$, output register r_o, and working registers $r_{m+1},\ldots,r_n$. Let $J_1,\ldots,J_e$ be the conditional instructions in P, listed in the order in which they occur in P. Construct a finite tree as follows.

Read the instructions in P. If H occurs before J_1 then there is only one node in the tree, assign H to that node. Otherwise assign 1 to the top node. In this case there are two nodes immediately below it. Where J_1 is the instruction if $S(r)$ then i else j, construct the left hand node as follows. From instruction i move downwards in P to the first conditional or halt instruction (which may be instruction i). If it is J_k then assign k, if it is an H assign H. A similar assignment is made for the right hand node, starting from the j-th instruction. In general, if H is assigned to a node then there are no nodes below it. Suppose k is assigned to a node. If k is also assigned to a node above this node then there are no nodes below. Otherwise there are two nodes immediately below, as described.

This is a finite tree, and it represents the various paths the machine can take through P. Next we will assign terms to each node in the tree, which will show what operations have been performed between conditionals. To do this we need $e+1$ lists of algebra variables, each of length n : $x_{oi},x_{1i},\ldots,x_{ni}$, $i=0,1,\ldots,e$.

Assignment to the top node: Assume $x_{oo},x_{1o},\ldots,x_{no}$ are the contents of the registers when the programme starts. Read the operational instructions down to the first conditional or halt. Write terms built up from the operation symbols, $x_{oo},\ldots,x_{no}$, and giving the contents of the registers at the first conditional or halt instruction. Assign the list of these n terms to the top node.

Suppose the first nonoperational instruction is conditional (J_1). Then there are two nodes just below. Assignment to the lefthand node: assume $x_{11},\ldots,x_{n1}$ are the contents of the registers, and the machine starts at instruction i. Write n terms which give the operations to the first conditional or halt instruction below i, and assign them to the lefthand node. (If instruction i is a conditional or halt then these n terms are $x_{o1},\ldots,x_{n1}$.) A similar assignment is made to the righthand node. If k is assigned to a node, and there are two nodes immediately below it, then similar assignments are made to these two nodes, with $x_{ok},\ldots,x_{nk}$ in the place of $x_{o1},\ldots,x_{n1}$.

In order to obtain a direct term which defines f we assign direct terms to each node in the tree, starting with the nodes at the bottom.

Suppose H is assigned to a bottom node, with $t_o,\ldots,t_n$ the list of terms assigned. Assign the direct term t_o (the contents of the output register) to this node. Suppose the number k is assigned to a bottom node. Then k is also assigned to a node above. This corresponds to a loop in the programme. Assign $p_k(t_o,\ldots,t_n)$ to this node, where p_k is an $(n+1)$-ary function variable, and $t_o,\ldots,t_n$ is the list of terms assigned to this node.

Suppose i is assigned to a node which is not a bottom node, and i is not assigned to a node below it. This node corresponds to the conditional instruction J_i : if S(r) then j else k. Assign the following direct term to this node: $DC_S(t,s_1,s_2)$, where t,s_1,s_2 are as follows. Let $t_o,\ldots,t_n$ be the list of terms assigned to this node. Let s_3,s_4 be the terms assigned to the two nodes immediately below, s_3 to the lefthand one, s_4 to the righthand one. r is the list of contents of the registers with numbers $i_1,i_2,\ldots,i_1$. Let t be the list of terms with numbers $i_1,i_2,\ldots,i_1$ from $t_o,\ldots,t_n$. s_1 is obtained from s_3 by replacing $x_{oi},\ldots,x_{ni}$ with $t_1,\ldots,t_n$. s_2 is obtained from s_4 in the same way.

Suppose k is assigned to a node, and k is also assigned to a node below it. This node corresponds to the conditional instruction J_k : if S(r) then * else **. Assign $FP[\lambda p_k,x_{1k},\ldots,x_{nk}.\ DC_S(x,s_3,s_4)]\ (t_o,\ldots,t_n)$ to this node, where $s_3,s_4,t_o,\ldots,t_n$ are as in the case above, the list x consists of elements numbered $i_1,i_2,\ldots,i_1$ from $x_{ok},\ldots,x_{nk}$.

In this way a direct term is assigned to each node in the tree. Let t be the direct term assigned to the top node. The free variables in t are among $x_{oo},\ldots,x_{no}$. Let t_o be obtained from t by replacing $x_{oo},x_{m+1,o},\ldots,x_{no}$ by $\underline{u}$. It remains to prove that t_o defines f, that is, for any $a_1,\ldots,a_m \in A$ $f(a_1,\ldots,a_m)$ = the value of t_o when the values of $x_{10},\ldots,x_{mo}$ are $a_1,\ldots,a_m$.

Let N be a node in the tree. Let $t_o,\ldots,t_n,t$ be the list of terms and the direct term assigned to N. Then the free algebra variables in $t_1,\ldots,t_n,t$ are among $x_{oi},\ldots,x_{ni}$ for some i.

The free function variables in t are among p_i. We will define a new programme P_N as follows. P_N has the same registers as P, all of which are regarded as input registers (in which u is an acceptable input). P_N begins with the operational instructions in P from which $t_o,\ldots,t_n$ were constructed and then follows the instructions in P below these, with the following replacements: if J_k is an instruction in P and the number k is assigned to a node above N, and to N or a node below N, then the new instruction in P_N is: $r_o := p(r_1,\ldots,r_n),H$.

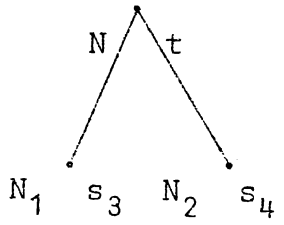
Note that if N is the top node then P_N and P are the same, with the exception that all registers are regarded as input registers in P_N.

Claim: Let N be a node in the tree, t the term assigned to N, p_i the free function variables in t, f_i any corresponding list of n-ary partial functions. Then the programme P_N and the term t define the same partial function, when p_i are interpreted by f_i.

The proof of this is by induction on the nodes in the tree, starting at the bottom and is trivial in all cases, except possibly when N is a node to which the number k has been assigned, and k is also assigned to a node below. In this case t is the term

$$FP[\lambda p_k, x_{ok}, x_{1k},\ldots,x_{nk}.\ DC_S(x,s_3,s_4)]\ (t_1,\ldots,t_n)$$

(see the corresponding case above). Let N_1 and N_2 be the two nodes immediately below N. Let P_1 and P_2 be the programme for N_1 and N_2 respectively.



By the induction hypothesis, P_1 defines the same partial function as s_3, P_2 defines the same partial function as s_4, for any choice of partial n-ary functions f_i. Let P' be the following programme, it has the same registers and instructions as P, with one difference. The first instruction for P' is new : go to J_k. This instruction is added in order to avoid the first operational instructions of P (which define $t_1,\ldots,t_n$) when P' starts. Let g^j, h^j, $j \in \omega$, be the following n-ary partial functions: $g^j(a_1,\ldots,a_n) \simeq b$ if the P' gives output b when loaded with $a_1,\ldots,a_n$, and the instruction J_k has been applied at most j times. h^o is the totally undefined function. h^{j+1} is the function defined by $DC_S(x,s_3,s_4)$ when P_k is interpreted as h^j. It is obvious that $g^j \leq g^{j+1}$. Let $g = \bigcup_{j\in\omega} g^j$, g is the function defined by P'. Then $h^j \leq h^{j+1}$. Let $h = \bigcup_{j\in\omega} h^j$. Then h is the function defined by $FP[\lambda p_k, x_{ok},\ldots,x_{nk}.\ DC_S^{j\in\omega}(x,s_3,s_4)]$. By induction on j one can prove that $g^j = h^j$ for all j.

Obviously $g^o = h^o$ as both are totally undefined. Suppose $g^j = h^j$. To prove that $g^{j+1} = h^{j+1}$ let us compute $g^{j+1}(a_1,\ldots,a_n)$ and $h^{j+1}(a_1,\ldots,a_n)$. Let a be the list of the elements with numbers $i_1,\ldots,i_l$ from $a_1,\ldots,a_n$. There are two cases : $S(a)$ and $\neg S(a)$. We take the first case only, as the second case is similar. Then $h^{j+1}(a_1,\ldots,a_n)$ is the value of s_3 with $a_1,\ldots,a_n$ for $x_{1k},\ldots,x_{nk}$, h^j for p_k. As noted above this is the output of P_1 with input $a_1,\ldots,a_n$, p_k interpreted as h^j. To find the value of $g^{j+1}(a_1,\ldots,a_n)$ load $a_1,\ldots,a_n$ into P' and let the machine run. First instruction: go to J_k. Second instruction: test whether or not $S(a)$. But as $S(a)$ P' runs as P_1, with one difference: if the instruction

goto J_k is met, then P_1 sets $r_o := h^j(r_1$ Suppose such an instruction is met. Then P' $g^j(r_1,\ldots,r_n) = c$ then P' will give output further applications of J_k. Hence the total n tions of J_k is a most j+1, and $g^{j+1}(a_1,\ldots,$ $g^j(r_1,\ldots,r_n) = h^j(r_1,\ldots,r_n)$, P_1 also gives o $h^{j+1}(a_1,\ldots,a_n) = c$. If $g^j(r_1,\ldots,r_n)$ is und either gives no output, or it gives an output a applications of J_k. In that case the total nu of J_k is more than j+1, hence $g^{j+1}(a_1,\ldots,a_$ P_1 gives no output as $h^j(r_1,\ldots,r_n)$ is undef $h^{j+1}(a_1,\ldots,a_n)$ is undefined. This proves tha

It follows that g = h, and P' defines th as $FP[\lambda p_k, x_{ok},\ldots,x_{nk}.\ DC_S(x,s_3,s_4)]$. From th to prove that P defines the same function as the claim and so the proposition.

For example, consider the following program register r_1, output register r_o, working regi

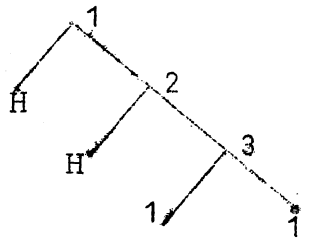
1. if $S(r_1)$ then 2 else 4 J_1
2. $r_o := r_1$
3. H
4. $r_2 := \sigma(r_1)$
5. if $S(r_2)$ then 6 else 8 J_2
6. $r_o := r_1$
7. H
8. $r_1 := \sigma(r_1)$
9. goto 1 J_3

It is easy to see that this programme computes the function

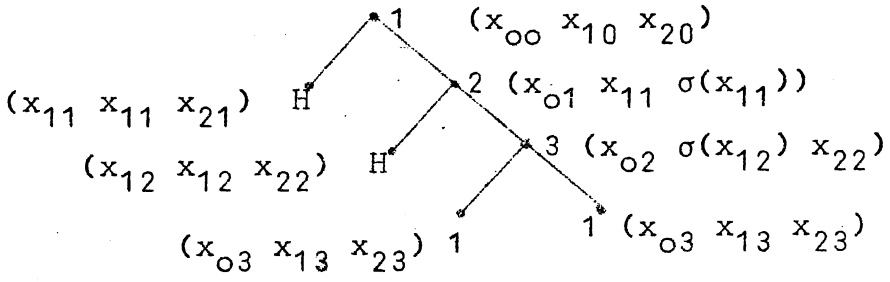
$$f(x) = FP[\lambda p', y.\underline{DC}_S(y, y, \underline{DC}_S(\underline{\sigma}(y), y, p\underline{\sigma}(y)))](x)$$

but consider the term manufactured by the argument for 3.1.

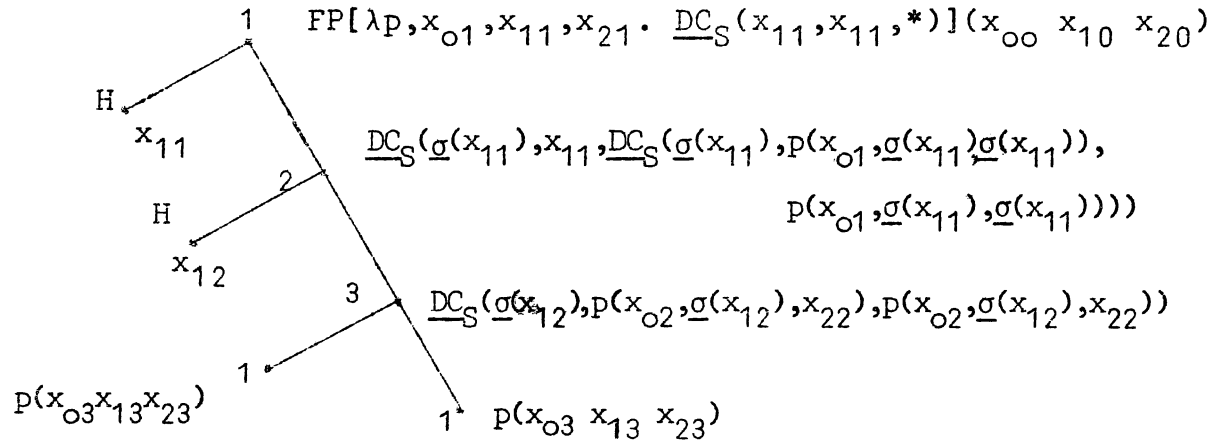
First, its conditionals J_1, J_2, J_3 are the instructions I_1, I_5, I_9 and its tree simply,



The following lists are assigned to the nodes:



The following terms are assigned to the nodes:



where * denotes the term at node 2. The final term is $FP[\lambda p, x_{01}, x_{11}, x_{21} \cdot \underline{DC}_S(x_{11}, x_{11}, *)]\ (\underline{u}, x_{10}, \underline{u})$. Clearly our construction of an induction term from a fap is inefficient.

3.2. Proposition. If a function is directly inductively definable then it is fap-computable.

We will prove a result which is slightly more general:
Claim: Let t be a direct term with free algebra variables among $x_1, \ldots, x_n$, free function variables p_i. Then there is a programme P with n input registers such that for any list of partial functions f_i, t and P define the same partial function with f_i in the place of p_i. If $p \in p_i$ then p can occur in an instruction for P as follows: $r_o := p(r)$, H, where r_o is the variable for the output register.

The proposition follows immediately from the claim. t will be a direct term with no free function variables.

Proof of the claim. This is by induction on the complexity of t which is one of the following.

(i) x_i , $1 \leq i \leq n$

(ii) $\underline{u}$

(iii) $\underline{\sigma}(t_1,\ldots,t_k)$

(iv) $\underline{DC_S}(t_1,\ldots,t_k,t_{k+1},t_{k+2})$

(v) $p(t_1,\ldots,t_k)$

(vi) $FP[\lambda p,y_1,\ldots,y_k.t_o](t_1,\ldots,t_k)$

where $t_o,\ldots,t_{k+2}$ are algebra terms. P will have $r_1,\ldots,r_n$ as input registers, r_o as output register, and the property that the contents of the input registers are not changed. We give the instruction for P in the cases (i), (ii), (iii) and (vi).

(i) $r_o := r_i$, H

(ii) H

(iii) By the induction hypothesis there are programmes $P_1,\ldots,P_k$ which define the same partial functions as $t_1,\ldots,t_k$. They all have the same input registers $r_1,\ldots,r_n$, and the same output register r_o. P wil have the following working registers: $r_{n+1},\ldots,r_{n+k}$, the working registers of $P_1,\ldots,P_k$. We assume that $r_{n+1},\ldots,r_{n+k}$ are not working registers in any of $P_1,\ldots,P_k$.
Instructions for P:

1. Instructions for P_1. Replace H by

   $r_{n+1} := r_o$
   goto 2

2. Instructions for P_2. Replace H by

   $r_{n+2} := r_o$
   goto 3
   .
   .
   .

k. Instructions for P_k. Replace H by

$r_{n+k} := r_o$

goto k+1

k+1 $r_o := \sigma(r_{n+1}, r_{n+2}, \ldots, r_{n+k})$

H

This programme computes the value of t_1 and puts it in r_{n+1}. Then it computes the value of t_2 and puts it in r_{n+2} and so on until it finally gives as output the value $\sigma(t_1, \ldots, t_k)$.

(vi) It suffices to construct a programme P' which defines the same partial function as $FP[\lambda p, y_1, \ldots, y_k . t_o]\ (z_1, \ldots, z_k)$. P can be obtained from P' by adding instructions for $t_1, \ldots, t_k$, as in case (iii). By the induction hypothesis there is a programme P_o which defines the same partial function as t_o, for any choice of partial functions p_i. Let $r_1, \ldots, r_n, r_{n+1}, \ldots, r_{n+k}$ be the input registers for P_o ($r_1, \ldots, r_n$ for $x_1, \ldots, x_n$, $r_{n+1}, \ldots, r_{n+k}$ for $y_1, \ldots, y_k$), with r_o the output register of P_o. P' will have the same input, output and working registers as P_o. The contents of the input registers are not changed during a computaion of P_o (by the induction hypothesis); the contents of the registers $r_{n+1}, \ldots, r_{n+k}$ may be changed during a computation of P'. The contents of $r_1, \ldots, r_n$ are not changed during a computation of P'. The input registers of P will be $r_1, \ldots, r_n$. The contents of these will not be changed during a computation of P.

The instructions for P' will be the same as the instructions for P_o, with the following change: replace $r_o := p(r)$, H by "put r into $r_{n+1}, \ldots, r_{n+k}$, goto the first instruction".

It remains to prove that P' and $t' = FP[\lambda p, y_1, \ldots, y_k . t_o]\ (z_1, \ldots, z_k)$ define the same (n+k)-ary partial function.

(The free variables in t' are $x_1,\ldots,x_n,z_1,\ldots$ — to prove that P' and t' define the same k-ar— for each choice of $x_1,\ldots,x_n$. So let $a_1,\ldots,a_$ $x_1,\ldots,x_n$ and place $a_1,\ldots,a_n$ into the regist— P', and of P_o. Define the k-ary functions g^j, follows: $g^j(b_1,\ldots,b_k) = c$ if P' gives outpu— $b_1,\ldots,b_k$ in $r_{n+1},\ldots,r_{n+k}$, and the first inst— applied at most j times. Obviously $g^j \leq g^{j+1}$ Let $g = \bigcup_{j\in\omega} g^j$. Then g is the function define— be totally undefined. Let h^{j+1} be the functio— with h^j in the place of p. By the induction — is the function which is defined by t_o when p as h^j. Then $h^j \leq h^{j+1}$. Let $h = \bigcup_{j\in\omega} h^j$. Then defined by t'. By induction on j one can pro— for all j. Hence $g = h$, and proposition 3.2 i

4. Theorem 2.

This is a term which is not direct

$$FP[\lambda p', y.\ \underline{DC}_S(y,y,\underline{DC}_S(\sigma_1(y),y,\sigma_2(p'(\sigma_1($$

and, in the way of illustration, here is an fapS function it defines. The programme has input re register r_o, and working registers r_2, r_3.

1. if $S(r_1)$ then 2 else 4
2. $r_o := r_1$
3. if $s = \emptyset$ then H else *
4. $r_2 := \sigma_1(r_1)$
5. if $S(r_2)$ then 6 else 8
6. $r_o := r_1$

7. if $s = \emptyset$ then H else *
8. $s := (1, r_o, r_1, r_2, r_3)$
9. $r_1 := \sigma_1(r_1)$
10. goto 1
11. * : $r_2 = r_o$
12. restore (r_o, r_1, r_3)
13. $r_3 := \sigma_1(r_1)$
14. $r_o := \sigma_2(r_2, r_3)$
15. if $s = \emptyset$ then H else *

There is a single block, instructions I_8 - I_{12}, with return instruction I_1.

4.1. Proposition. If a function is inductively definable then it is fapS-computable.

Proof: It is sufficient to prove the following result. If t is an algebraic term with free function variables $p_1, \ldots, p_e$ then there is a programme P, involving a stack, in which operational instructions $r_j := p_i(\tau)$, $1 \leq i \leq e$, are allowed, such that t and P define the same partial function for any substitution of $p_1, \ldots, p_e$. Here τ is a list of operational terms or polynomials in the programming language with indeterminates the register variables so that the instruction $r_j := p_i(\tau)$ abbreviates several operational instructions (excluding the halt) followed by an application of p_i.

This is proved by induction on the complexity of a term t which has one of the following forms:

(i) x_i, $1 \leq i \leq m$

(ii) $\underline{u}$

(iii) $\underline{\sigma}(t_1,\ldots,t_k)$

(iv) $\underline{DC}_S(t_1,\ldots,t_k,t_{k+1},t_{k+2})$

(v) $p(t_1,\ldots,t_k)$

(vi) $FP[\lambda p,y_1,\ldots,y_k.t_o]\ (t_1,\ldots,t_k)$

We will construct a fapS for t from programmes for its subterms. There are six cases, the cases (ii), (iii) and (vi) are given below. Assume that the free algebraic variables of t are among $x_1,\ldots,x_m$ (i.e. the partial function defined by t is m-ary). Let $r_1,\ldots,r_n$ be the input registers of P.

Case (ii). Let r_o be the output register. The programme of P has only one instruction: if $s = \emptyset$ then H else H.

Case (iii). Let r_o be the output register, let $r_{m+1},\ldots,r_k$ be working registers. By the induction hypothesis there are programmes $P_1,\ldots,P_k$ for the terms $t_1,\ldots,t_k$. By convention $r_1,\ldots,r_m$ are the input registers, r_o the output register, for all of them. Assume further that $r_{m+1},\ldots,r_{m+k}$ are not working registers for any of them. P is this.

```
    ⌜ s := (1;r_o,...,r_n)
    | goto 1 →
    | *r_{m+1} := r_o
    ⌞ restore (r_o,...,r_m,r_{m+1},...,r_n)
      goto 2
1 → instructions for P_1
2   ⌜ s := (2;r_o,...,r_n)
    | goto 2 →
    | *r_{m+2} := r_o
    ⌞ restore (r_o,...,r_{m+1},r_{m+3},...,r_n)
      goto 3
```

```
   2 → instructions for P_2
   3⌈
    |
    |
    |
    ⌊
    .
    .
    .
   k⌈ s : = (k;r_o,...,r_n)
    | goto k →
    | *r_{m+k} : = r_o
    | restore (r_o,...,r_{m+k-1},r_{m+k},...,r_n)
    ⌊
     goto k+1
   k → instructions for P_k
 k+1  r_o : = σ(r_{m+1},...,r_{m+k})
      if  s = ∅  then  H  else *
```

The programme does this: when $a_1,\ldots,a_m$ are loaded into the input registers. It stacks $(1,u,a_1,\ldots,a_m,y,\ldots,u)$, then acts as P_1. If P_1 gives no output neither will P. Suppose P_1 gives output b_1. When one comes to the final instruction where P_1 halts then $(1,u,a_1,\ldots,a_m,u,\ldots,u)$ is still in the stack. So it goes to * in the first block, sets $r_{m+1} = b_1$, returns $u,a_1,\ldots,a_m,u,\ldots,u$ to the registers excluding u to r_{m+1}. The stack is now empty. It then goes to 2, and so on. If $P_1,\ldots,P_k$ give outputs $b_1,\ldots,b_k$ then P will put these values into $r_{m+1},\ldots,r_{m+k}$, and finally give output $\sigma(b_1,\ldots,b_k)$.

Case (vi). t is $FP[\lambda p,y_1,\ldots,y_k.t_o](t_1,\ldots,t_k)$. It suffices to construct a fapS P' for the term $t' = FP[\lambda p,y_1,\ldots,y_k.t_o](z_1,\ldots,z_k)$. A programme for t can easily be constructed from P' by adding the first part of the programme in (iii) - the instructions above k+1 - to the programme of P'.

The free algebraic variables in t_o are among $x_1,\ldots,x_m$, $y_1,\ldots,y_k$, the free function variables among $p,p_1,\ldots,p_e$. By the induction hypothesis there is a fapS P which defines the same partial function as t_o for any substitution of $p,p_1,\ldots,p_e$. Let the input registers of P_o be $r_1,\ldots,r_n$ (for $x_1,\ldots,x_m$) and $r_{m+1},\ldots,r_{m+k}$ (for $y_1,\ldots,y_k$), output register r_o. P' will have the same input, output and working registers as P_o. The programme of P' is obtained from the programme of P_o by replacing each instruction $r_j := p(\tau)$ with a new block:

$$\left\lceil\begin{array}{l} s := (i;r_o,\ldots,r_n) \\ r_{m+1},\ldots,r_{m+k} := \tau \\ \text{goto } i\rightarrow \\ *r_j := r_o \\ \text{restore } (r_1,\ldots,r_{j-1},r_{j+1},\ldots,r_n) \end{array}\right.$$

where i is a label not used for any other block and $i\rightarrow$ denotes the first instruction in P_o.

It remains to prove that t' and P' define the same partial function for any choice of $p_1,\ldots,p_e$.

The free algebraic variables in t' are $x_1,\ldots,x_m,z_1,\ldots,z_k$. Fix $a_1,\ldots,a_m$. Define k-ary partial functions f^l, $l\in\omega$, as follows: f^o is totally undefined, f^{l+1} is the function defined by t_o with f^l stubstituted for p, and $x_i = a_i (1\leq i\leq m)$. Now $f^l \leq f^{l+1}$, so let $f = \bigcup_{l\in\omega} f^l$; f is the function defined by t' with $a_1,\ldots,a_m$ fixed.

Without loss of generality, we can assume that the contents of the registers $r_1,\ldots,r_m$ are not changed when P_o performs a computation. Define k-ary partial functions g^l, $l\in\omega$, as follows: $g^l(b_1,\ldots,b_k) = c$ if P' gives output c when $a_1,\ldots,a_m,b_1,\ldots,b_k$ are loaded into $r_1,\ldots,r_m,r_{m+1},\ldots,r_{m+k}$, and at no stage there are

as many as l tuples from the new blocks (i.e. the blocks in P' which are not in P_o) in the stack. Then $g^l \leq g^{l+1}$. Let $g = \bigcup_{l\in\omega} g^l$. Then g is the function defined by P' with $r_i = a_i (1 \leq i \leq m)$. It suffices to prove that $f^l = g^l$ for all l.

This is done by induction. The basis $l=0$ is obvious. Suppose $f^l = g^l$. By the first induction hypothesis f^{l+1} is defined by P_o with g^l in the place of p and $a_1,\ldots,a_m$ in $r_1,\ldots,r_m$. Let p be f^l $(=g^l)$. Load $a_1,\ldots,a_m,b_1,\ldots,b_k$ into the input registers of P_o and P'. The two machines do the same operations until P_o meets an instruction $r_j := p(\tau)$, then P' meets the i-th block. P_o will set $r_j = f^l(\tau)$ $(=g^l(\tau))$. P' will stack $(i,r_o,\ldots,r_n)$, then it sets $r_{m+1},\ldots,r_{m+k} = \tau$ and goes to $i \rightarrow$.

If $g^l(\tau)$ is not defined then P_o sets $r_j = u$, that is, P_o gives no output and $f^{l+1}(b_1,\ldots,b_k)$ is undefined. If $a_1,\ldots,a_m,\tau$ is placed into the input registers of P' then P' either gives no output, or there is a stage with l tuples from the new blocks in the stack. With $a_1,\ldots,a_m,b_1,\ldots,b_k$ in the input registers P' either gives no output, or there is a stage with $l+1$ tuples from the new blocks in the stack. Hence $g^{l+1}(b_1,\ldots,b_k)$ is undefined.

Suppose $g^l(\tau) = c$, where $c \neq u$. Then P_o sets $r_j = c$, and continues. If $a_1,\ldots,a_m,\tau$ is loaded into the input registers of P' it will give output c, and at no stage is there l tuples from the new blocks in the stack. With $a_1,\ldots,a_m,b_1,\ldots,b_k$ as input the following will happen. P' stacks $(i,r_o,\ldots,r_n)$. At a later stage $(i,r_o,\ldots,r_n)$ is taken out of the stack, r_j is set equal to c. Until now the number of tuples from the new blocks in the stack has been at most l. At this point the contents of

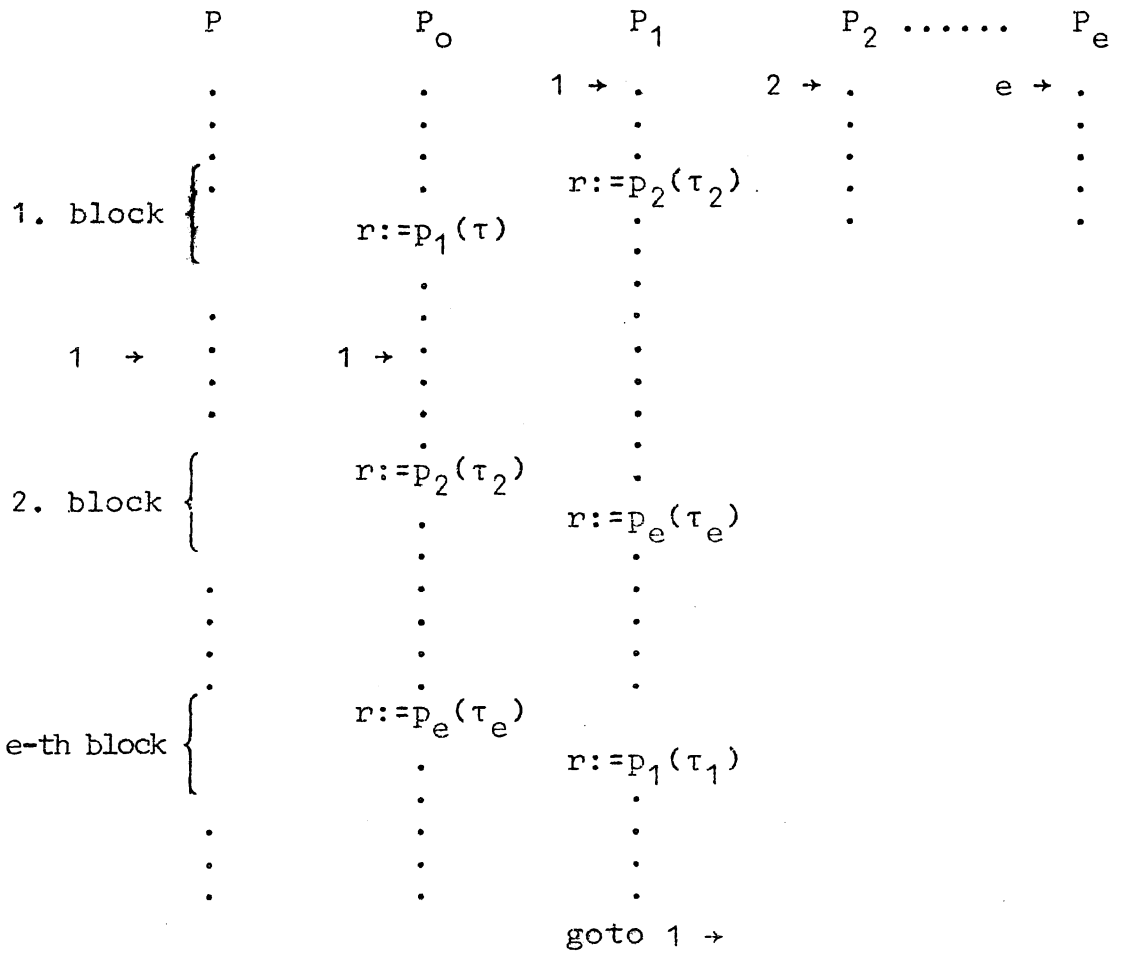
the registers are the same for P_o and P', and they will do the same operations until the next occasion an instruction $r_j := p(\tau)$ is met. This proves that $f^{l+1} = g^{l+1}$.

Q.E.D.

4.2. Proposition. If a function is fapS-computable then it is inductively definable.

Proof: Let P be a fapS with input registers $r_1,\ldots,r_m$, output register r_o and working registers $r_{m+1},\ldots,r_n$.

Let e be the number of stacking blocks in P. Here is a diagram of P and of some other programmes.



$1 \rightarrow$ is a mark for the return instruction of the first block, $2 \rightarrow$ for the second block, and so on. P_o is obtained from P by replacing the i-th block with $r_j := p_i(\tau_i)$, $i=1,\ldots,e$ and each halting block with H. p_i is an n+1-ary function variable, τ_i is the list of n+1 terms which is given by the operational instructions in the i-th block. P_1 is a rearrangement of P_o. It begins with the return instruction for the first block, then come the instructions which are below this in P_o, and then the instructions in the first part of P_o, preceeding the return instruction. At the end is added goto $1 \rightarrow$. P_2 is a similar rearrangement of P_o, starting with $2 \rightarrow$, and so on.

$P_o, P_1, \ldots, P_e$ are programmes for register machines without stacks including indeterminate functions $p_1, \ldots, p_e$ over the structure. They each have registers $r_o, \ldots, r_n$, with output register r_o. P_o has input registers $r_1, \ldots, r_m$ but we take all registers as input registers in the other programmes. By 3.1 there are terms $t_o, t_1, \ldots, t_e$ which define the same partial functions as $P_o, P_1, \ldots, P_e$ for any choice of n-ary functions $p_1, \ldots, p_e$. In t_o we will replace $p_1, \ldots, p_e$ by some terms so to obtain a term for P. From t_1 we will define a term t_{11} which will be substituted for p_o in $t_o, t_2, \ldots, t_e$, and hence obtain $t_{o1}, t_{21}, \ldots, t_{e1}$ in which p_1 is not free. From t_{21} we define a term t_{22} which will be substituted for p_2 in $t_{o1}, t_{31}, \ldots, t_{e1}$ and thus obtain $t_{o2}, t_{32}, \ldots, t_{e2}$ in which p_1, p_2 are not free, etc. Suppose we have obtained terms $t_{oi}, t_{i+1,i}, \ldots, t_{ei}$ in which $p_1, \ldots, p_i$ are not free. Let $t_{i+1,i+1}$ be $FP[\lambda p_{i+1}, x_1, \ldots, x_n \cdot t_{i+1,i}]$, and substitute $t_{i+1,i+1}$ for p_{i+1} in $t_{o1}, t_{i+2,i}, \ldots, t_{ei}$. The terms thus obtained are denoted $t_{o,i+1}, t_{i+2,i+1}, \ldots, t_{e,i+1}$. Let t be t_{oe}.

It remains to prove that t defines the same function as P. First define a series of programmes as follows: Let P_{jo} be P_j,

$0 \leq j \leq e$. Suppose $P_{jo}, P_{j,1}, \ldots, P_{ji}$ are defined, $0 \leq j \leq e$. Then $P_{j,i+1}$ is defined from $P_{j,i}$ by replacing $r := p_{i+1}(\tau_{i+1})$ with the (i+1)-th block. If $i=0$ we also replace H with if $S = \emptyset$ then H else *. Note that $P = P_{oe}$. It is sufficient to prove the following claim.

Claim: t_{ij} and P_{ji} define the same function for any substitution of $p_{i+1}, \ldots, p_e$, where $j=0,i,i+1,\ldots,e$. The proof is by induction on i. Obviously true for $i=0$. Suppose the claim is true for i. To prove that it is true for i+1 we first show that $t_{i+1,i+1}$ and $P_{i+1,i+1}$ define the same partial function for any choice of $p_{i+2}, \ldots, p_e$. Given such a choice, define f^k, $k \in \omega$, as follows: f^o is totally undefined. f^{k+1} is the function defined by $t_{i+1,i}$ with f^k in the place of p_{i+1}. Then $f^k \leq f^{k+1}$. Let $f = \bigcup_{k\in\omega} f^k$. f is the function defined by $t_{i+1,i+1}$. Define g^k, $k \in \omega$, as follows: $g^k(a_o,\ldots,a_n) = b$ if $P_{i+1,i+1}$ gives output b when $a_o,\ldots,a_n$ are loaded into its input registers, and at no stage are there as many as k tuples in the stack with number i+1. Then $g^k \leq g^{k+1}$. Let $g = \bigcup_{k\in\omega} g^k$. g is the function defined by $P_{i+1,i+1}$. To prove that f=g it suffices to prove $f^k = g^k$ for all k. This is by induction on k.

Case $k=0$ is trivial. Suppose $f^k = g^k$. By the first induction hypothesis $t_{i+1,i}$ and $P_{i+1,i}$ define the same function for any substitution of p_{i+1}, in particular for the choice $p_{i+1} = f^k$. Hence f^{k+1} is defined by $P_{i+1,i}$ when $p_{i+1} = f^k$. Load $a_o,\ldots,a_n$ into the input registers of $P_{i+1,i}$ and $P_{i+1,i+1}$. The two programmes coincide until $P_{i+1,i}$ comes to the instruction $r_j := p_{i+1}(\tau_{i+1})$. Then $P_{i+1,i+1}$ comes to the (i+1)th block. $P_{i+1,i}$ sets $r_j = f^k(\tau_{i+1})$, which by the second induction hypothesis is the same as $g^k(\tau_{i+1})$. $P_{i+1,i+1}$ stacks $(i+1,r_o,\ldots,r_n)$

sets $r_o,\ldots,r_n = \tau_{i+1}$ and goes to $i+1 \rightarrow$, the first instruction in the programme. If $g^k(\tau_{i+1})$ is undefined then $P_{i+1,i}$ gives no output, i.e. $f^{k+1}(a_o,\ldots,a_n)$ is undefined. If τ_{i+1} is loaded into the input registers of $P_{i+1,i+1}$ then either it yields no output, or at some stage there are k tuples with number $i+1$ in the stack (since $g^k(\tau_{i+1})$ is undefined). Hence $P_{i+1,i+1}$ either gives no output when applied to $a_o,\ldots,a_n$, or at some stage there are $k+1$ tuples with number $i+1$ in the stack. Hence $g^{k+1}(a_o,\ldots,a_n)$ is undefined. If $g^k(\tau_{i+1}) = b$ then $P_{i+1,i}$ sets $r_j = b$, and goes on. If τ_{i+1} is loaded into the registers $P_{i+1,i+1}$ it will give output b, and at no stage is there as many as k tuples with number i in the stack. Hence the following happens after $(i+1,r_o,\ldots,r_n)$ is stacked: $P_{i+1,i+1}$ meets if $S = \emptyset$ then H else * with b in the output register, and $(i+1,r_o,\ldots,r_n)$ will be that tuple in the stack is topmost. Then it goes to * in the $(i+1)$-th block, sets $r_j = b$ and sets $r_o,\ldots,r_{j-1},r_{j+1},\ldots,r_n$ equal to the values they had when $(i+1,r_o,\ldots,r_n)$ was stacked. Hence $P_{i+1,i}$ and $P_{i+1,i+1}$ have the same contents in the registers, and will do the same operations until $r_j := p_{i+1}(\tau_{i+1})$ is met. Note that during these operations the number of tuples with number $i+1$ in the stack is at most k. So $P_{i+1,i}$ gives output c iff $P_{i+1,i+1}$ gives output c, and at no stage there is more than k tuples with number $i+1$ in the stack. Hence $f^{k+1}(a_o,\ldots,a_n) = g^{k+1}(a_o,\ldots,a_n)$, that is, $f^{k+1} = g^{k+1}$.

It remains to prove that $t_{j,i+1}$ and $P_{j,i+1}$ define the same function for $j=0,i+2,i+3,\ldots,e$. By the induction hypothesis t_{ji} and P_{ji} define the same function for any choice of p_{i+1}, in particular for the function defined by $t_{i+1,i+1}$, denoted h.

It suffices to prove that $P_{j,i}$ (with $p_{i+1}=h$) and $P_{j,i+1}$ define the same function. Apply both programmes to $a_o,\ldots,a_n$. They perform the same operations until $P_{j,i}$ meets the instruction $r_k := p_{i+1}(\tau_{i+1})$. Then $P_{j,i+1}$ meets the i+1-st block. $P_{j,i}$ sets $r_k = h(\tau_{i+1})$. $P_{j,i+1}$ stacks $(i+1,r_o,\ldots,r_n)$, sets $r_o,\ldots,r_n = \tau_{i+1}$ and goes to $i + 1 \rightarrow$. Then it will do the same operations as $P_{i+1,i+1}$ (with τ_{i+1} as input) until eventually $(i+1,r_o,\ldots,r_n)$ is taken out of the stack again, in which case r_k is set equal to the output of $P_{i+1,i+1}$ which is $h(\tau_{i+1})$, and $r_o,\ldots,r_{k-1},r_{k+1},\ldots,r_n$ take the values they had when $(i+1,r_o,\ldots,r_n)$ was put in the stack. Then the contents of the registers and of the stack of $P_{j,i}$ and $P_{j,i+1}$ are the same, and they will do the same operations until $r_k := p_{i+1}(\tau_{i+1})$ is met. This proves the claim and 4.2.

Q.E.D.

References

[1] R.C. Constable & D. Gries. On classes of program schemata. SIAM Journal on Computing 1. (1972) pp.66-118.

[2] S. Feferman Inductive schemata and recursively continuous functionals. pp.373-392 of R.O. Gandy & J.M.E. Hyland (eds.) Logic colloquium '76, North-Holland, Amsterdam, 1977.

[3] J.E. Fenstad On axiomatising recursion theory pp.385-404 of J.E. Fenstad & P.G. Hinman (eds.). Generalized recursion theory. North-Holland, Amsterdam, 1975.

[4] J.E. Fenstad Computation theories: an axiomatic approach to recursion on general structures, pp.143-168 of G. Müller, A. Oberschelfp & K. Potthoff (eds.). Logic conference, Kiel 1974, Springer-Verlag, Heidelberg, 1975.

[5] J.E. Fenstad On the foundation of general recursion theory: Computations versus inductive definability. pp.99-111 of J.E. Fenstad, R.O. Gandy, & G.E. Sacks: Generalized recursion theory II, North-Holland, Amsterdam, 1978.

[6] J.E. Fenstad Recursion theory: an axiomatic approach. Springer-Verlag, Berlin, to appear.

[7] H. Friedman Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory, pp.316-389 of R.O. Gandy & C.M.E. Yates (eds.) Logic colloquium '69, North-Holland, Amsterdam, 1971.

[8] S.C. Kleene — Recursive functionals and quantifiers of finite type I. Transactions American Mathematical Society 91 (1959) pp.1-52.

[9] S.C. Kleene — Recursive functionals and quantifiers of finite type II. Transactions American Mathematical Society 108 (1963) pp.106-142.

[10] A.I. Mal'cev — Algebraic systems. Springer-Verlag, Berlin, 1973.

[11] Z. Manna & A. Shamir — The convergence of functions to fixed-points of recursive definitions. Theoretical Computer Science 6 (1978) pp.109-141.

[12] J. Moldestad — Computations in higher types. Springer-Verlag, Berlin, 1977.

[13] J. Moldestad, V. Stoltenberg-Hansen & J.V. Tucker — Finite algorithmic procedures and computation theories. Matematisk institutt, Universitetet i Oslo, Preprint Series, Oslo, 1978.

[14] Y.N. Moschovakis — Abstract first-order computability I. Transactions American Mathematical Society 138 (1969), pp.427-464.

[15] Y.N. Moschovakis — Abstract first-order computability II. Transactions American Mathematical Society 138, (1969) pp.465-504.

[16] Y.N. Moschovakis — Axioms for computations theories - first draft pp.119-255 of R.O. Gandy & C.M.E. Yates (eds.) Logic colloquium '69, North-Holland, Amsterdam, 1971.

[17] R.A. Platek — Foundations of recursion theory. Ph.D. Thesis, Stanford University, Stanford, 1966.

[18] J.C. Shepherdson — Computations over abstract structures: serial and parallel procedures and Friedman's effective definitional schemes, pp.445-513 of H.E. Rose & J.C. Shepherdson (eds.) Logic colloquium '73. North-Holland, Amsterdam, 1975.

[19] J.V. Tucker — Computing in algebraic systems. Matematisk institutt, Universitetet i Oslo, Preprint Series, Oslo, 1978.

[20] J.V. Tucker — On the function theory of algebraic systems. In preparation.